情報処理学会論文誌 Vol.53 No.2 601–611 (Feb. 2012)

プログラム依存グラフを用いた増分的なコードクローン検出

肥後 芳樹[1,a)] 植田 泰士[1,b)] 西野 稔[1,c)] 楠本 真二[1,d)]

受付日 2011年6月14日, 採録日 2011年11月7日

概要：近年，増分的なコードクローン検出法が注目されている．増分的な検出法では，検出結果（もしくはその中間生成物）はデータベースに保存され，次回以降の検出に利用される．このため，同じファイル群から何度もコードクローン検出を行う場合では，2 回目以降の検出時間を大幅に短縮することができる．本論文では，プログラム依存グラフを用いた増分的な検出法を提案する．プログラム依存グラフを用いた検出は，行単位の検出などの他の検出法に比べて検出に長い時間を必要とするため，増分的な検出を行うメリットは大きい．また，提案手法に基づいてツールを開発し，適用実験を行った．実験の結果，提案手法は，従来のプログラム依存グラフを用いた検出法とほぼ同じコードクローンを検出するが，検出時間は大幅に短くなっていることを確認した．

キーワード：コードクローン，プログラム依存グラフ

Incremental Code Clone Detection Using Program Dependency Graph

Yoshiki Higo[1,a)] Yasushi Ueda[1,b)] Minoru Nishino[1,c)] Shinji Kusumoto[1,d)]

Received: June 14, 2011, Accepted: November 7, 2011

Abstract: Incremental code clone detection attracts much attention in the last few years. In incremental detections, code clone detection results (or their intermediate products) persist by using databases, and they are used in next code clone detection. However, no incremental detection technique has been proposed for PDG-based detection, which requires much more time to detect code clones than line- or token-based detection. In this paper, we propose a PDG-based incremental code clone detection technique for improving practicality of PDG-based detection. A prototype tool has been developed based on the proposed method, and it has been applied to open source software. We confirmed that detection time extremely shortened and its detection result is almost the same as one of an existing PDG-based detection technique.

Keywords: code clone, program dependency graph

1. はじめに

コードクローンの存在はソフトウェアの保守作業に悪影響を与える要因の 1 つであるといわれている．コードクローンとは，ソースコード中に存在する重複コードであり，コピーアンドペーストなどのさまざまな理由により生成される [7]．バグが顕在化した場合，ソースコード中の該当部分に対して修正が加えられる．しかし，そのコードクローンがシステム中に存在した場合は，それらに対しても同様の修正を検討する必要がある．修正を行った保守管理者が，コードクローンの存在に気づかなかった場合，システムには潜在的なバグが残ってしまう可能性がある．

このような，コードクローンに起因する問題を解決・軽減するための 1 つの方法として，リファクタリングがあげられる．リファクタリングを行うことにより，コードクローン部分は 1 つの関数やメソッドにまとめられるため，今後の修正作業により，その部分には不整合が起こらない．

これまでに，ソースコード中からコードクローンを自動的に検出するさまざまな手法が提案されている [1], [7]．既

[1] 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University, Suita, Osaka 565–0871, Japan
[a)] higo@ist.osaka-u.ac.jp
[b)] yss-ueda@ist.osaka-u.ac.jp
[c)] m-nisino@ist.osaka-u.ac.jp
[d)] kusumoto@ist.osaka-u.ac.jp

存の検出技術は，用いている技術により，行単位での検出，字句単位での検出，抽象構文木を用いた検出，プログラム依存グラフを用いた検出，メトリクスやフィンガプリントなどを用いた検出に大別される．これらの中で，抽象構文木を用いた検出とプログラム依存グラフを用いた検出はリファクタリングに向いているといわれている [14]．しかし，これらの検出技術は，他の検出技術に比べ検出に必要な計算量が高く，コードクローン情報が欲しいときに，瞬時にその情報を得ることが難しい．

また，近年，増分的なコードクローン検出[*1]が注目を集めている．増分的なコードクローン検出とは，検出結果とその中間生成物をデータベースなどに保管し，次回以降の検出時に利用する検出手法である．検出対象ファイルが前回の検出時から更新されていない場合は，そのファイル自身は解析されず，ファイルに関する必要な情報はデータベースから取得される．このような枠組みを用いることによって，同じファイル集合から何度も検出を行う場合は，2 回目以降の検出時間を大幅に短縮することができる．

増分的なコードクローン検出手法は，行単位での検出と字句単位での検出を行う手法がこれまでに提案されている [5], [9]．しかし，リファクタリングなどに有効な，抽象構文木を用いた検出やプログラム依存グラフを用いた検出はまだ提案されていない．また，これらの手法は，検出に必要な計算コストが高いため，増分的に検出を行うメリットは大きい．このような現在の状況をふまえ，本論文では，プログラム依存グラフを用いた増分的なコードクローン検出手法を提案する．

2. プログラム依存グラフ

プログラム依存グラフ（Program Dependency Graph, 以降 PDG）とは，ソースコード中の要素（文や条件節の式）間の依存関係を表現した有向グラフである．PDG のノードは要素，エッジは要素間の依存関係を表す．一般的な PDG では，依存関係は**制御依存**と**データ依存**の 2 種類が存在する．本論文では，PDG はメソッド単位で構築されるものとする．

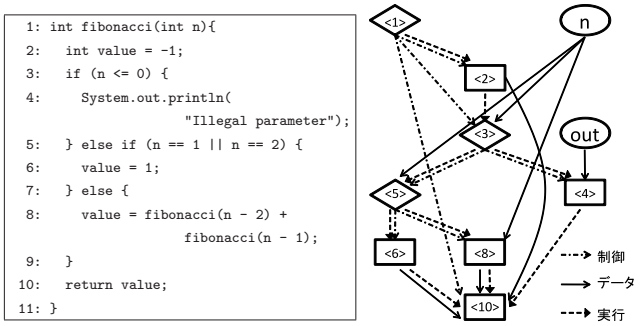
以下のすべての条件を満たすとき，要素 s_1 と s_2 の間には制御依存関係が存在する．

- s_1 は条件節の式である．
- s_1 の結果により，s_2 が実行されるか否かが決定する．

また，以下のすべての条件を満たすとき，要素 s_3 と s_4 の間にはデータ依存関係が存在する．

- s_3 では，変数 v を定義している（変数 v に対して代入処理を行っている）．
- s_4 では，変数 v を参照している．
- s_3 から s_4 への経路の中に，変数 v を再定義しないものが存在する．

[*1] Incremental Code Clone Detection



(a) フィボナッチ数列を計算するメソッド　(b) 生成される PDG

図 1 PDG の例
Fig. 1 PDG example.

本研究では，上記の依存関係に加えて，**実行依存**も用いる．実行依存とは実行の順序を表す依存関係であり，要素 s_5 が実行された直後に要素 s_6 が実行される可能性がある場合，s_5 から s_6 へ実行依存エッジが引かれる．実行依存を PDG に追加することにより，コードクローンの検出能力が高まることが確認されている [6]．

図 1 は，フィボナッチ数列の計算を行うプログラムとその PDG である．PDG のノードの数字は，その要素のソースコード中での位置（行番号）を表している．なお，`<1>` のノードは，メソッドの入り口ノードを表し，そのノードは便宜上，条件節の式と見なされる．また，`n` や `out` は，それぞれ仮引数や共有変数を表すノードである．

3. 提案手法

図 2 は提案手法の概観を表している．提案手法は**解析処理**と**検出処理**の 2 つからなる．解析処理では，ソースコードから PDG が構築され，その情報がデータベースに保存される．検出処理では，データベースから PDG の情報を取り出し，コードクローンの検出を行う．提案手法は，コードクローン検出に必要な PDG の情報を再利用することにより，ファイル読み込み，ソースコード解析，および PDG の構築に必要なコストを削減する．その結果，コードクローン検出を行う場合に，より短時間で結果を得ることができる．つまり，提案手法は，多くの PDG 情報を再利用できる状況において有効である．提案手法が有効な状況としては，たとえば下記のものがあげられる．

- 複数のリビジョンから順にコードクローンを検出する．
- 着目するファイルに関連するコードクローンのみを検出する．

5 章では，これらの状況を想定した提案手法の評価を行っている．

以降，本章では，3.1 節で提案手法で利用する種々の定義を行い，3.2 節では解析処理について説明する．その後，3.3 節で検出処理について説明する．

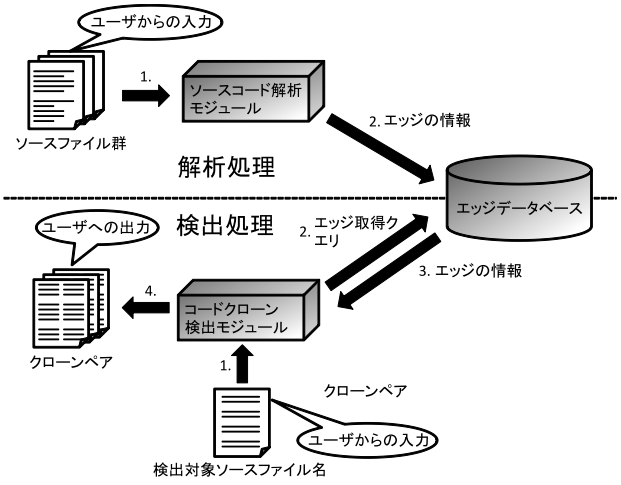


図 2 提案手法の概観
Fig. 2 Overview of the proposed method.

3.1 定義

まず，プログラム依存グラフを定義する．

定義 1（プログラム依存グラフ） 本研究で用いるプログラム依存グラフは連結グラフであるため，そのグラフ（g とする）を構成するエッジ集合として定義することができる．

$$g := \{e_1, e_2, \cdots, e_n\}$$

また，本論文では，エッジを下記のように定義する．

定義 2（エッジ） エッジ e の出発ノードを v_1，到達ノードを v_2，エッジの種類（データ・制御・実行）を t とすると，エッジ e は下記の式で表される．

$$e = (v_1, v_2, t)$$

次に，エッジの**隣接関係**を定義する．

定義 3（隣接関係） エッジ $e_1 = (v_1, v_2, t_1)$，エッジ $e_2 = (v_3, v_4, t_2)$ とすると，これら 2 つのエッジの隣接関係は下記の式により定義される．

$$\begin{aligned} &incident(e_1, e_2) \\ &\quad := (v_1 = v_3 \wedge v_2 \neq v_4) \vee (v_1 = v_4 \wedge v_2 \neq v_3) \\ &\qquad \vee (v_2 = v_4 \wedge v_1 \neq v_3) \vee (v_2 = v_3 \wedge v_1 \neq v_4) \end{aligned}$$

つまり，エッジ e_1 と e_2 が 1 つのノードを共有していれば，$incident(e_1, e_2)$ は真となる．

次に，隣接関係を用いて**エッジ間の経路**を定義する．

定義 4（エッジ間の経路） 連結グラフであるため[*2]，任意のエッジ e_i とエッジ e_j の間には経路が存在する．経路が複数存在する場合もあるため，エッジ e_i と e_j の間の経路集合を，$PATHS(e_i, e_j)$ と表す．なお，ここでの経路は閉路を含まないものとする．閉路を含まない経路とは，その経路上に存在するすべてのノードを 1 度しかたどらないものである．

[*2] 厳密には，参照されていない引数があると孤立ノードが存在するが，そのような孤立ノードは無視する．

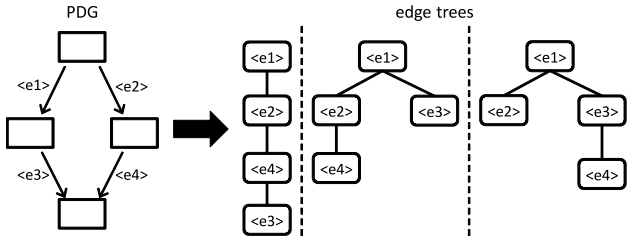


図 3 1 つの PDG から生成されるエッジツリー
Fig. 3 Edge trees generated from a PDG.

次に，**エッジツリー**という新しいデータ構造を定義する．

定義 5（エッジツリー） エッジツリーのノードは，プログラム依存グラフ $g = \{e_1, e_2, \cdots, e_n\}$ を構成するエッジであり，ツリーのエッジはグラフのエッジ間の隣接関係である．また，エッジツリーのノード数はグラフのエッジ数と等しい（グラフ中のすべてのエッジをノードとして持つ）．木であるため閉路は含まない．つまり，PDG が閉路を含む場合は，1 つのエッジツリーに，PDG エッジの隣接関係がすべて現れることはない．

図 3 はエッジツリーの簡単な例を表している．この図に示すように，1 つの PDG から複数のエッジツリーが生成されうる．図 3 はエッジ e_1 を根とする 3 つのエッジツリーを表している．このように 1 つのエッジツリーがすべての隣接関係を表現しているわけではない．

次に，**エッジツリーの集合**を定義する．

定義 6（エッジツリーの集合） $TREES(g)$ は，プログラム依存グラフ g から生成されうるすべてのエッジツリーの集合である．

また，ここで，**エッジツリー上の経路**を定義する．

定義 7（エッジツリー上の経路） $treepath(t, e)$ は，エッジツリー t において，根からノード e までの経路である．

定義 4 と定義 7 を用いると，定理 1 は自明である．

定理 1 e_r をエッジツリー t の根とすると，下記の式が成り立つ．

$$treepath(t, e) \in PATH(e_r, e)$$

また，連結グラフでは，$PATH(e_r, e) \neq \phi$ はつねに真となる．つまり，あるエッジから他の任意のエッジへの経路が少なくとも 1 つ存在する．つまり，連結グラフからは少なくとも 1 つのエッジツリーが生成される．

次に，**グラフエッジの同値関係**を定義する．

定義 8（グラフエッジの同値関係） 2 つのエッジ $e_1 = (v_1, v_2, t_1)$ と $e_2 = (v_3, v_4, t_2)$ の同値関係は，下記の式で定義される．

$$e_1 \equiv e_2 := (t_1 \equiv t_2) \wedge (v_1 \equiv v_3) \wedge (v_2 \equiv v_4)$$

e_1 と e_2 の依存関係の種類が等しいとき $t_1 \equiv t_2$ は真とな

る．$v_1 \equiv v_2$ は，コードクローン検出の状況に応じた定義をすべきである．たとえば，完全に字面が等しいコード片のみをコードクローンとして検出したい場合は，ノード v_1 と v_2 を表す文字列を直接比較すべきである．また，ある程度の差違も許容して検出したい場合は，何らかの正規化処理を行った後に，比較すべきである．

定義 8 を用いて，ツリーノード間の経路の同値関係を定義する．

定義 9（ツリーノード間の経路の同値関係） 2 つのツリーノード間の経路 $p_1 = (e_1, e_2, \cdots, e_n)$ と $p_2 = (f_1, f_2, \cdots, f_m)$ において，その同値関係は下記の式で定義される．

$$p_1 \equiv p_2 := (|p_1| = |p_2|) \wedge \forall i (e_i \equiv f_i)$$

つまり，ツリーノード間の経路 p_1 と p_2 が同数のノードを持ち，各ノードのペアが同値であれば，p_1 と p_2 は同値である．

次に，**エッジツリーの同値関係**を定義する．

定義 10（エッジツリーの同値関係） 2 つのエッジツリー $t_1 \in TREES(g_1)$ と $t_2 \in TREES(g_2)$ において，その同値関係は下記の式で定義される．なお，グラフ g のエッジ数を $|g|$ と表す．

$$\begin{aligned} & t_1 \equiv t_2 \\ & := (|g_1| = |g_2|) \wedge \exists (e_1, e_2, \cdots, e_{|g_1|}) \exists (f_1, f_2, \cdots, f_{|g_2|}) \\ & \quad \forall k \left(\bigcup_{1 \le i \le |g_1|} \{e_i\} = g_1 \wedge \bigcup_{1 \le j \le |g_2|} \{w_j\} \right. \\ & \quad \left. = g_2 \wedge treepath(t_1, e_k) \equiv treepath(t_2, f_k) \right) \end{aligned}$$

つまり，エッジツリー t_1 と t_2 はエッジの数が等しく，t_1 の任意のノードから根への経路と同値関係にある経路が t_2 に存在しており，かつ t_2 の任意のノードから根への経路と同時関係にある経路が t_1 に存在している場合に，同値となる．

上記の定義を用いることによって，クローンペアは次のように定義される．

定義 11（クローンペア） プログラム依存グラフ g_1 と g_2 上のサブグラフ s_1 と s_2 において，下記の式が真になる場合に，s_1 と s_2 はクローンペアとなる．

$$\begin{aligned} & clonepair(s_1, s_2) := (s_1 \cap s_2 = \phi) \\ & \wedge \exists t_1 \exists t_2 (t_1 \in TREES(s_1) \wedge t_2 \in TREES(s_2) \wedge t_1 \equiv t_2) \end{aligned}$$

定義 11 のクローンペアの中には出力する必要のないものが含まれている．そのようなコードクローンを取り除いた定義を以下に示す．

定義 12（出力されるクローンペア）

$$\begin{aligned} & outputclonepair(s_1, s_2) := clonepair(s_1, s_2) \\ & \quad \wedge \neg \exists (s_1', s_2') (clonepair(s_1', s_2') \wedge s_1 \subset s_1' \wedge s_2 \subset s_2') \end{aligned}$$

提案手法では，$outputclonepair$ を満たすクローンペアが出力される．

3.2 解析処理

解析処理の入力と出力を以下に示す．

入力 ソースファイル群

出力 更新されたソースファイル内のグラフエッジ群

図 2 の上部は解析処理の概要を表している．解析処理では，以下の手順で行われる．

1. 各ソースファイルの更新時刻と，前回解析処理を行った時刻と比較する．前回の解析処理時刻よりも新しい更新時刻を持つファイルを**更新されたファイル**と呼ぶ．この比較により，更新されたファイル群を得る．
2. 解析モジュールは更新されたファイル群から PDG を構築し，それに含まれるエッジの情報をエッジデータベースに保存する．

手順 2. では，すでにそのファイルの古いバージョンの情報が保存されている場合は，その情報を削除したのち，新しいバージョンの情報が追加される．

3.3 検出処理

検出処理の入力と出力を以下に示す．

入力 コードクローン検出対象ソースファイル名

出力 入力として与えられたファイルに関連するクローンペアの集合

図 2 の下部は検出処理の概要を表している．検出処理は下記の手順で行われる．

1. ユーザはコードクローンを検出したいファイルを指定する．
2. 検出モジュールはそのファイル名をクエリとしてエッジデータベースに問い合わせる．
3. エッジデータベースは，クエリのファイル内に存在するエッジと等価なエッジ群を返す．
4. 検出モジュールは，エッジデータベースより返されたエッジ群から，クローンペアを構築しユーザに提示する．

検出処理を実現するためには，エッジ群からクローンペアを生成するアルゴリズムが必要である．まず，メソッドを入力として，そのメソッドと関連のあるクローンペアの集合を出力するアルゴリズムを Algorithm 1 に示す．なお，あるメソッドと関連のあるクローンペアとは，そのクローンペアのコードクローンのうち少なくとも一方がそのメソッド内に存在しているクローンペアである．

アルゴリズムの主要な部分を説明する．

Algorithm 1 detect (m)

Input: m: メソッド
Output: C: メソッド m に関連するクローンペアの集合
1: $C \leftarrow \phi$
2: **for all** e_1 such that $e_1 \in m$ **do**
3: **for all** e_2 such that $e_2 \equiv e_1 \wedge e_2 \neq e_1$ **do**
4: $C \leftarrow C \cup create(e_1, e_2, \phi)$
5: **end for**
6: **end for**
7: **return** C

1 行目 出力するクローンペアの集合を保存する変数 C を空集合で初期化する．
2 行目 入力として与えられたメソッド m に含まれるエッジを 1 つずつ取ってくる（このエッジを e_1 とする）．なお，この処理は，メソッド ID を元にしたデータベースへの問合せで実現している．入力として与えられたメソッドのソースコードが解析されるわけではない．
3 行目 データベース検索を用いて，エッジ e_1 と等価なエッジ（このエッジを e_2 とする）を 1 つずつ取ってくる．
4 行目 エッジ e_1 と e_2 を引数として，アルゴリズム *create* を用いてクローンペア情報を構築し，C に加える．

Algorithm 2 create (e_1, e_2, U)

Input: e_1, e_2: エッジのペア，U: すでにチェックしたエッジの集合
Output: (S_1, S_2): $u_1 \in S_1$ かつ $u_2 \in S_2$ である *outputclonepair*
1: $(S_1, S_2) \leftarrow (\{e_1\}, \{e_2\})$
2: $U \leftarrow U \cup \{e_1, e_2\}$
3: **for all** e_x such that $incident(e_1, e_x) \wedge e_x \notin U$ **do**
4: **for all** e_y such that $e_x \equiv e_y \wedge incident(e_2, e_y) \wedge e_y \notin U$ **do**
5: $(S_x, S_y) \leftarrow create(e_x, e_y, U)$
6: $(S_1, S_2) \leftarrow (S_1 \cap S_x, S_2 \cap S_y)$
7: **end for**
8: **end for**
9: **return** (S_1, S_2)

次にアルゴリズム *create*（Algorithm 2）を説明する．このアルゴリズムは，エッジのペアを入力として受け取り，そのエッジを含むクローンペアを出力する．以下に *create* アルゴリズムの主要名部分を説明する．
1 行目 引数で与えられた与えられたエッジのペア（e_1 と e_2）を出力するクローンペアの要素（S_1 と S_2）とする．
2 行目 引数で与えられた 2 つのエッジをチェック済みエッジ集合に加える．
3 行目 e_1 と隣接関係にあり，またチェックされていないエッジを 1 つずつ取ってくる（このエッジを e_x とする）．
4 行目 e_2 と隣接関係にあり，まだチェックされておらず，e_x と等価なエッジを 1 つずつ取ってくる（このエッジを e_y とする）．
5 行目 e_x，e_y，およびこれまでにチェックしたエッジの集合を引数として，アルゴリズム *create* を再起呼び出しする．
6 行目 5 行目の結果得られた，クローンペアを成すエッジの集合を S_1 と S_2 に加える．

4. 実装

提案手法をツールとして実装した．規模は約 10,000 行であり，PDG の構築に必要なソースコード解析には，著者らの研究グループで開発している MASU [13] を利用した．現在のところ，対象言語は Java のみであるが，ソースコードから PDG を構築するモジュールを実装すれば，他のプログラミング言語に対しても適用することができる．また，データベースには SQLite を利用している．

4.1 PDG ノードの同値関係

提案手法では，PDG ノードの同値関係については定義していないため，実装したツールでは，以下の定義を用いている．

定義 13（PDG ノードの同値関係） s_1 と s_2 をプログラムの要素とし，v_1 と v_2 はそれらを表す PDG のノードとすると，同値関係は以下の式で定義される．

$$v_1 \equiv v_2 := normalize(s_1) = normalize(s_2)$$

normalize は正規化処理を行う関数であり，変数とリテラルを下記のルールで正規化を行う．
変数 変数ごとに特殊な字句が用意される．型が同じであっても違う変数は違う字句に置換される．
リテラル リテラルの型ごとに特殊な字句が用意される．型が同じであれば，値が違っていても同じ字句に置換される．

4.2 データベース構成

ツールのデータベースには 5 つのテーブルが存在している．**表 1** にそれらを示す．解析処理で取得したエッジは，エッジテーブルに登録される．各エッジにつき，そのエッジを含むメソッド ID，そのハッシュ値，始点の頂点 ID，終点の頂点 ID を登録する．検出処理では，このハッシュ値を用いてデータベースからの等価なエッジの取得処理が行われる．実装したツールでは，ハッシュ値を求めるアルゴリズムとして MD5 を用いている．定義 2 に示すように，本論文では，エッジ e は (v_1, v_2, t) で表現している．よって，e のハッシュ値は v_1，v_2，および t を結合したバイト列の MD5 値として求める．始点ノードと終点ノードは ID が登録される．これは，同一のノードを始点および終点として持つエッジが複数あっても，データベースサイズを大きくしないための工夫である．ノードの情報はノードテーブルに保存されており，ノード ID をキーとしてその情報を取得できる．

表 1 データベースのテーブル一覧
Table 1 Tables in database.

テーブル名	登録内容
エッジ	メソッド ID，ハッシュ値，始点のノード ID，終点のノード ID
ノード	ノード ID，メソッド ID，開始位置，終了位置
ファイル	ファイル ID，ファイル名（ファイルへの絶対パス），更新時刻
メソッド	メソッド ID，メソッド名
対応関係	ファイル ID，メソッド ID

また，ファイルの情報を登録するためのテーブルがあり，ファイル ID，ファイル名（ファイルの絶対パス），更新時刻が登録される．このテーブルを用いることにより，更新されたファイルのファイル ID を取得する．

ファイルとメソッドの対応関係（どのファイルの中にどのメソッドが存在しているか）は，対応関係テーブルに保存される．一部のファイルが更新された場合には，ファイルテーブルと対応関係テーブルを用いて，更新されたファイルにあるメソッドの ID を取得し，それを用いてエッジテーブルに登録されている更新されたファイルの古いエッジ情報を削除する．

5. 評価

本章では，提案手法を評価するために行った実験について述べる．この実験は以下の 2 つの実験からなる．

実験 A 提案手法の検出速度を評価するための実験．効率的にコードクローンが検出できているのかどうかを確認する．

実験 B 提案手法の検出精度を評価するための実験．PDG を用いた検出法として，検出結果が適切であるかどうかを確認する．

なお，この実験は，下記に示す仕様のワークステーションを用いて行った：

CPU Intel Xeon E5405（quad-core, 2.0 GHz）
Memory 8 GB
OS Windows 7 Enterprise（64 bit）

実験対象は，Ant である．Ant を実験対象とした理由を下記に示す．

- Ant は Java 言語を用いて開発されている．現在のところ，提案手法を実装したツールが対応しているのは Java 言語のみである．
- Ant の最新リビジョンの規模が約 20 万行あり，また開発期間も数年と実験対象規模として十分なため．

5.1 実験 A：検出時間の評価

この実験では，下記の 2 つの項目について評価を行った．

項目 1 Ant のすべてのリビジョンに対して順にコードクローン検出を行い，その時間を測定した．このような検出処理は，コードクローンに関係する研究で頻繁に行われている．近年，コードクローンの存在がどの程度ソフトウェア保守作業を阻害しているのかを調査する試みが行われている [8], [11], [12]．また，コードクローンの履歴情報を可視化する手法も提案されている [2]．このような研究では，増分的な検出法を用いることによって，効率的に検出処理を行える．

項目 2 あるコード片からバグを検出した場合は，そのコード片とコードクローンになっている部分についても同様のバグがないかを調査することが望ましい．本項目ではそのような状況を想定して，ある 1 つのファイルが指定されたときに，それとコードクローンになっている部分を検出するのに必要な時間を測定する．

5.1.1 項目 1 の評価

この項目の評価は下記の手順で行われた．

STEP1 最初のリビジョンをチェックアウトする．
STEP2 最初のリビジョンからコードクローンを検出する．この STEP では，すべてのソースファイルに対して解析処理と検出処理が実行される．
STEP3 次のリビジョンをチェックアウトする．
STEP4 STEP3 でチェックアウトしたリビジョンからコードクローン検出を行う．この STEP では，アップデートされたファイルについてのみ解析処理が実行され，すべてのファイルに対して検出処理が実行される．
STEP5 もし次のリビジョンが存在すれば，STEP3 に戻る．

この評価では，STEP2 と STEP4 の合計時間を計測した．その結果，5,903 リビジョンから 15 時間 2 分でコードクローン検出を完了できた．リビジョンごとの検出時間を**図 4** に示す．また，PDG を用いたコードクローン検出ツール Scorpio [6] を利用して，増分的でない検出の時間を測定した．増分的でない検出では，STEP2 と STEP4 において，そのリビジョンに含まれるすべてのソースファイルをから Scorpio を用いてコードクローン検出を行った．約 3,300 リビジョンを 1 日で解析できたが，システムが大きくなるに従って検出時間も長くなり，すべてのリビジョンからの検出を終えるのに約 4 日半を要した．

すべてのリビジョンに対して，その前のリビジョンとの間で更新されたファイル数を調べたところ，平均値が 3.53，中央値が 1，最大値が 733 であった．つまり，多くのリビ

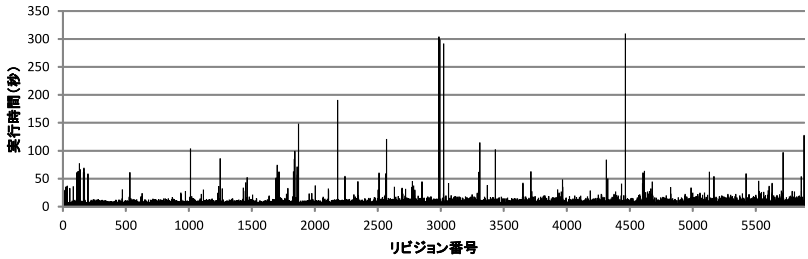


図 4 Ant の各リビジョンからコードクローンを検出するのに必要な時間
Fig. 4 Elapsed time to detection for every revision.

表 2 項目 2 の実行時間
Table 2 Detection time in context 2.

処理	STEP1	STEP3		
		平均値	中央値	最大値
実行時間	3 分 18 秒	2.9 秒	2.2 秒	12.8 秒

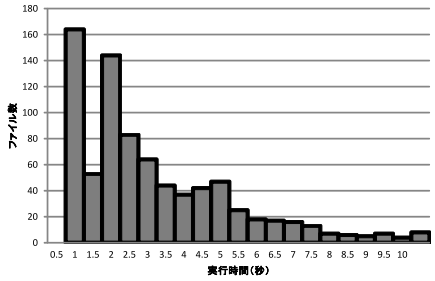


図 5 各ファイルに対する検出時間の分布
Fig. 5 Frequency table of detection time for each file.

ジョンでは，少数のファイルのみが更新されており，残りの更新されていないファイルについてはエッジの情報が再利用されている．この再利用により，ファイル読み込み，ソースコード解析，PDG 構築のコストが抑えられ，増分的でない検出との間で検出時間に差がでた．今回実験対象とした Ant のように，リビジョン間での更新されたファイル数が全ファイル数に対して非常に少ない場合には，大きな速度向上を期待できる．

5.1.2 項目 2 の評価

項目 2 の評価は，Ant の最新リビジョン（リビジョン番号：1,075,177，ファイル数：804，行数：203,580）に対して，下記の手順で行った．

STEP1 すべてのファイルに対して解析処理を実行．
STEP2 各ファイルに対して検出処理を実行．

表 2 は，STEP1 と STEP2 に要した時間を表している．STEP1 ではデータベースを作成するために，3 分 18 秒が必要であった．STEP2 では，単一ファイルに対する検出処理は最大でも 12.8 秒であり，項目 2 の状況で実用的に利用可能な検出速度といえる．

図 5 は，各ファイルに対する検出時間のヒストグラムを表している．X 軸は，500 ミリ秒ごとに区切られており，検出時間が $500n$ ミリ秒以上 $500(n+1)$ ミリ秒未満であったファイルが，左から n 番目の区域に入る．つまり，n 番目の区域の棒グラフの値は，検出時間が $500n$ ミリ秒以上 $500(n+1)$ ミリ秒未満であったファイルの数である．たとえば，ラベルが 1 の区域は，検出時間が 500 ミリ秒以上 1 秒未満であったファイルの数を表している．この図より，多くのファイルに対する検出処理が数秒以内で終わっていることが分かる．つまり，ユーザは，着目しているファイルを検出対象として指定すると，その数秒後にはそのファイルに関連するコードクローン群を得ることができる．Scorpio を用いた場合には検出に数分を必要とするため，提案手法を用いることにより，検出の高速化が実現されていることが分かる．

5.2 実験 B：検出精度の評価

検出精度の評価では，既存の PDG を用いた検出ツールである Scorpio [6] と検出結果の比較を行った．この評価でも，実験 A の項目 1 と同様に，Ant の最新リビジョンを検出対象とした．比較対象として Scorpio を用いたのは以下の 2 つの理由のためである．

- Scorpio は提案手法と同じく実行依存付きの PDG を用いて検出処理を行うため，同型部分グラフをコードクローンとして検出する方法と，提案手法の検出法の差違を適切に比較することができる．
- Scorpio は著者らの研究グループで開発されたツールであり，検出の設定方法を熟知しているため，提案手法となるべく同じ設定にして検出処理を行うことができる．

また，提案手法と Scorpio が検出したコードクローンが一致するかどうかの判定には，Bellon らが提案した good 値と ok 値 [4] を用いた．good 値と ok 値の定義については付録を参照されたい．good 値と ok はいずれも 2 つのクローンペアのオーバラップの程度を 0%〜100%の割合で表す指標である．これらの値を用いることにより，提案手法により検出されたクローンペアと Scorpio により検出されたクローンペアのオーバラップの程度を計測し，閾値以上であれば，同一のクローンペアであると見なした．この実験では，これらの値の閾値として 70%を用いた．

804 個のソースファイルから，提案手法は 831 個，Scorpio

```
    50:  e = replyToList.elements();
    51:  while (e.hasMoreElements()) {
    52:    mailMessage.replyto(e.nextElement().toString());
    53:  }
+   54:  e = toList.elements();
+   55:  while (e.hasMoreElements()) {
+   56:    String to = e.nextElement().toString();
    57:    try {
    58:      mailMessage.to(to);
+   59:      atLeastOneRcptReached = true;
+   60:    } catch (IOException ex) {
+   61:      badRecipient(to, ex);
    62:    }
    63:  }
++- 64:  e = ccList.elements();
++- 65:  while (e.hasMoreElements()) {
++- 66:    String to = e.nextElement().toString();
    67:    try {
    68:      mailMessage.cc(to);
++- 69:      atLeastOneRcptReached = true;
 +- 70:    } catch (IOException ex) {
 +- 71:      badRecipient(to, ex);
    72:    }
    73:  }
 + -74:  e = bccList.elements();
 + -75:  while (e.hasMoreElements()) {
 + -76:    String to = e.nextElement().toString();
    77:    try {
    78:      mailMessage.bcc(to);
 + -79:      atLeastOneRcptReached = true;
   -80:    } catch (IOException ex) {
   -81:      badRecipient(to, ex);
    82:    }
    83:  }
```

図 6 提案手法と Scorpio で一致しなかったコードクローンの例
Fig. 6 Mismatched code clone between the proposed method and Scorpio.

表 3 提案手法の再現率と適合率
Table 3 Precision and recall of the proposed method.

評価項目	*good*	*ok*
適合率	0.922	0.970
再現率	0.990	1.000

は 724 個のクローンペアを検出した．**表 3** は，Scorpio が検出したクローンペアの集合を正解集合とした場合の，提案手法の適合率と再現率を表している．Scorpio が検出したクローンペアの集合を $S_{scorpio}$，提案手法の検出結果 R に含まれるクローンペアの集合を $S(R)$，ok 値を用いて抽出した共通のクローンペアの集合を $S(R) \cap^{ok} S_{scorpio}$，good 値を用いて抽出したクローンペアの集合を $S(R) \cap^{good} S_{scorpio}$ とすると，再現率は下記の式で表される．

$$Recall_{ok}(R) := \frac{|S(R) \cap^{ok} S_{scorpio}|}{|S_{scorpio}|},$$

$$Recall_{good}(R) := \frac{|S(R) \cap^{good} S_{scorpio}|}{|S_{scorpio}|} \quad (1)$$

また，適合率は，下記の式で表される．

$$Precision_{ok}(R) := \frac{|S(R) \cap^{ok} S_{scorpio}|}{|S(R)|},$$

$$Precision_{good}(R) := \frac{|S(R) \cap^{good} S_{scorpio}|}{|S(R)|} \quad (2)$$

なお，$|A|$ は集合 A に含まれるクローンペアの数を表す．つまり，再現率は Scorpio が検出したクローンペアをどの程度提案手法も検出したのかを表し，適合率は提案手法が見つけたクローンペアのうちどの程度が Scorpio によっても検出されていたのかを表す．表 3 より，適合率と再現率はともに 1 に近い値であり，提案手法の検出結果は Scorpio の検出結果と近いことが分かる．つまり，提案手法は従来の PDG を用いた検出法と検出結果がきわめて似ている．

実験 A と実験 B の結果より，提案手法は従来の PDG を

用いた検出法と検出結果はほとんど変わらないが，検出に必要な時間を大幅に短縮できているといえる．

図 6 は，提案手法と Scorpio で一致しなかったクローンペアの例である．行頭の “+” は，その要素が提案手法によって検出されたコードクローンに含まれてることを表しており，“–” は，その要素が Scorpio によって検出されたコードクローンに含まれていることを表している．たとえば，64 行目は提案手法によって検出されたクローンペアの両方のコードクローンと，Scorpio によって検出されたクローンペアの片方のコードクローンに含まれている．64，65，66，69 行目は，提案手法によって検出されたクローンペアの両方のコードクローンに含まれている．その 2 つのコードクローンは，PDG 上ではノードを共有してはいるが，エッジは共有していない．提案手法のクローンペアの定義では，2 つのコードクローンがノードを共有することは禁止していないので，このようなコードクローンが検出される．一方，Scorpio は，コードクローンがノードを共有することを禁止している．このような定義の違いにより，図 6 に示すような一致しないコードクローンを検出される[*3]．

6. 関連研究

Hummel らは，各行の内容と位置情報を蓄積して，同内容の行を検索することでコードクローンを検出する手法を提案し，検出ツール ConQAT に実装している [9]．この手法では，まずソースコード中の各行に対して，変数名などに対する置換を行ったうえで，ハッシュ値を計算する．そして，そのハッシュ値とファイル名や行番号を組にした情報を蓄積する．ある行に対してコードクローンとなる行は，この情報をその行のハッシュ値を用いて検索することで，取得することができる．検出結果として出力するのは複数行からなるコードクローンであるが，それについてはあるコード片の各行とコードクローンになる行の集合から容易に構築することができる．提案手法は，Hummel らの手法と類似度が高い．異なる点は，データベースに登録する単位とその単位からコードクローンを構築する方法である．Hummel らの手法では，プログラムは行単位でデータベースに登録されており，その情報から連続した行をコードクローンとして検出する．一方，提案手法は，プログラムは PDG のエッジ単位でデータベースに登録されており，その情報から，同型な木構造を構築し，それをコードクローンとして検出する．手法の枠組み（検出の流れ）は，Hummel らが提案しているものと同様であり，その点については新規性はない．提案手法の新規性は，等価なエッジ群から木構造を構築するアルゴリズムである．しかし，このアルゴリズムにより，連続[*4]および非連続コードクローン[*5]を高速に検出できるようになった．Hummel らの手法は非連続コードクローンを検出することができない．

Göde らは，接尾辞木を改良し，情報の追加や削除が容易な汎用接尾辞木というデータ構造を定義して，それを用いた字句単位のコードクローン検出手法を提案している [5]．接尾辞木とは，文字列に対して定義される文字数と等しい個数の葉を持つ木構造であり，根からそれぞれの葉 i への経路が文字列の i 文字目から最後の文字までの部分文字列（接尾部）に対応する．接尾辞木を用いることで，文字列中の繰返し部分を高速に検出することができる．コードクローン検出においては，コードクローンがソースコード中の繰返し部分であることから，プログラム全体を 1 つの文字列として接尾辞木を構築し，検出を行う手法が用いられている [3], [10]．汎用接尾辞木は，1 つの文字列ではなく文字列の集合に対して定義され，根から葉への経路は各文字列における接尾部に対応する．各ファイルをそれぞれ 1 つの文字列として汎用接尾辞木を構築することにより，ファイルの追加や削除が容易になり，増分的な検出が可能になる一方，コードクローン検出の精度には影響を与えないことを示している．

7. おわりに

本論文では，PDG を用いた増分的なコードクローン検出手法を提案した．また，提案手法を実装したプロトタイプを紹介し，それを用いて行った実験についても述べた．この実験により，提案手法は，開発履歴の全リビジョンからのコードクローン検出を想定した実験では，従来の約 14%の時間で検出を完了することができ，バグを発見したファイルとのコードクローン検出を想定した実験では，平均 2.9 秒で検出処理を終えることができた．また，提案手法と増分的な検出法ではない従来手法との検出結果の比較では，従来手法の検出結果を正解集合とした場合の，提案手法の検出結果は再現率が 99%以上，適合率が 92%以上となり，提案手法は従来手法とほとんど結果が変わらないことを示した．

今後は，さらに検出時間を短縮するために手法を改良する予定である．現在は，コードクローンの検出は，対象ファイルを与えることで行っているが，メソッドを与えることでも検出ができるように改良を行う．バグ修正などのソースコード変更では，1 つのソースファイル全体にわたって改変が加えられる場合よりも，その中の一部のメソッドのみが改変される場合が多いと著者らは考えている．メソッド単位での検出を実現することによって，前回までの検出

[*3] ここでは，一致しないコードクローンを紹介したのみであり，どちらのコードクローンがより適切かという議論はしていない．

[*4] contiguous code clones．フォーマットの違い（改行位置，空白やタブの有無など）や字句単位の違い（変数名やリテラルの違いなど）のみを含むコードクローンを指す．

[*5] non-contiguous code clones. 字句単位よりも大きな違い（対応する文がない，文の順序が異なるなど）を含むコードクローン．

結果を再利用できる割合が増えるため，より高速に検出処理を行うことができる．

謝辞 本研究は一部，文部科学省「次世代 IT 基盤構築のための研究開発」（研究開発領域名：ソフトウェア構築状況の可視化技術の開発普及）の委託に基づいて行われた．また，日本学術振興会科学研究費補助金基盤研究（A）（課題番号：21240002）および萌芽研究（課題番号：23650014），文部科学省科学研究費補助金若手研究（B）（課題番号：22700031）の助成を得た．

参考文献

[1] Clone Detection Literature, available from ⟨http://www.cis.uab.edu/tairasr/clones/literature/⟩.
[2] Adar, E. and Kim, M.: Visualization and Exploration of Code Clones in Context, *Proc. 29th International Conference on Software Engineering*, pp.762–766 (2007).
[3] Baker, B.S.: Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance, *SIAM Journal on Computing*, Vol.26, No.5, pp.1343–1362 (1997).
[4] Bellon, S., Koschke, R., Antoniol, G., Krinke, J. and Merlo, E.: Comparison and Evaluation of Clone Detection Tools, *IEEE Trans. Softw. Eng.*, Vol.31, No.10, pp.804–818 (2007).
[5] Göde, N. and Koschke, R.: Incremental Clone Detection, *Proc. 13th European Conference on Software Maintenance and Reengineering* (2009).
[6] 肥後芳樹，楠本真二：プログラム依存グラフを用いたコードクローン検出法の改善と評価，情報処理学会論文誌，Vol.51, No.12, pp.2149–2168 (2010).
[7] 肥後芳樹，楠本真二，井上克郎：コードクローン検出とその関連技術，電子情報通信学会論文誌 D，Vol.J91-D, No.6, pp.1465–1481 (2008).
[8] Hotta, K., Sano, Y., Higo, Y. and Kusumoto, S.: Is Duplicate Code More Frequently Modified than Non-duplicate Code in Software Evolution?: An Emprical Study on Open Source Software, *Proc. 4th International Joint ERCIM/IWPSE Symposium on Software Evolution*, pp.73–82 (2010).
[9] Hummel, B., Juergens, E., Heinemann, L. and Conradt, M.: Index-Based Code Clone Detection: Incremental, Distributed, Scalable, *Proc. 26th IEEE International Conference on Software Maintenance*, pp.1–9 (2010).
[10] Kamiya, T., Kusumoto, S. and Inoue, K.: CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code, *IEEE Trans. Softw. Eng.*, Vol.28, No.7, pp.654–670 (2002).
[11] Krinke, J.: Is Cloned Code more stable than Non-Cloned Code?, *Proc. 8th IEEE International Working Conference on Source Code Analysis and Manupulation*, pp.57–66 (2008).
[12] Lozano, A. and Wermelinger, M.: Assessing the effect of clones on changeability, *Proc. 24th International Conference on Software Maintenance*, pp.227–236 (2008).
[13] 三宅達也，肥後芳樹，楠本真二，井上克郎：多言語対応メトリクス計測プラグイン開発基盤 MASU の開発，電子情報通信学会論文誌 D，Vol.J92-D, No.9, pp.1518–1531 (2009).
[14] Rysselberghe, F. and Demeyer, S.: Evaluating Clone Detection Techniques from a Refactoring Perspective, *Proc. 19th IEEE International Conference on Automated Software Engineering*, pp.336–339 (2004).

付　　録

定義 1 2 つのコード片（f_1 と f_2）の重なりの程度を以下の式で定義する．なお，$elements(f)$ はコード片 f に含まれるプログラム要素の集合を表す．

$$overlap(f_1, f_2) := \frac{|elements(f_1) \cap elements(f_2)|}{|elements(f_1) \cup elements(f_2)|} \tag{A.1}$$

定義 2 あるコード片（f_1）が他のコード片（f_2）に含まれている程度を以下の式で定義する．

$$contain(f_1, f_2) := \frac{|elements(f_1) \cap elements(f_2)|}{|elements(f_1)|} \tag{A.2}$$

定義 3 2 つのクローンペア（p_1 と p_2）の good 値は以下の式で定義される．

$$good(p_1, p_2) := min(overlap(p_1.f_1, p_2.f_1), \\ overlap(p_1.f_2, p_2.f_2)) \tag{A.3}$$

図 A·1 には 2 つのクローンペア（p_1 と p_2）が存在している．この場合，good 値は，

$$good(p_1, p_2) = min\left(\frac{5}{8}, \frac{6}{8}\right) = \frac{5}{8}$$

となる．閾値が 0.7 の場合，$\frac{5}{8} \leq 0.7$ であるため，2 つのクローンペアは一致しない．

定義 4 2 つのクローンペア（p_1 と p_2）の ok 値は以下の式で定義される．

$$ok(CP_1, CP_2) \\ := min(max(contained(CP_1.CF_1, CP_2.CF_1), \\ contained(CP_2.CF_1, CP_1.CF_1)), \\ max(contained(CP_1.CF_2, CP_2.CF_2), \\ contained(CP_2.CF_2, CP_1.CF_2))) \tag{A.4}$$

図 A·1 の例を用いた場合，ok 値は，

$$ok(p_1, p_2) = min\left(max\left(\frac{5}{6}, \frac{5}{7}\right), max\left(\frac{6}{6}, \frac{6}{8}\right)\right) = \frac{5}{6}$$

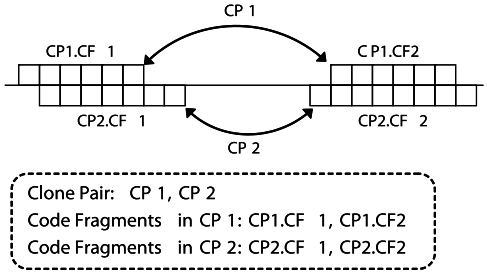


図 A·1 good 値と ok 値の算出例
Fig. A·1 Sample calculation of good and ok values.

となる．閾値が 0.7 の場合，$0.7 \leq \frac{5}{6}$ であるため，2 つのクローンペアは一致する．

肥後 芳樹 （正会員）

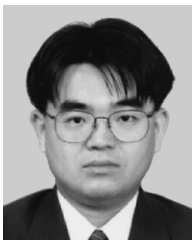
平成 14 年大阪大学基礎工学部情報科学科中退．平成 18 年同大学大学院博士後期課程修了．平成 19 年同大学院情報科学研究科コンピュータサイエンス専攻助教．博士（情報科学）．ソースコード分析，特にコードクローン分析やリファクタリング支援に関する研究に従事．電子情報通信学会，日本ソフトウェア科学会，IEEE 各会員．

植田 泰士

平成 13 年大阪大学基礎工学部情報科学科卒業．平成 15 年同大学大学院博士前期課程修了．現在，宇宙航空研究開発機構所属．在学中，コードクローン分析の研究に従事．

西野 稔

平成 21 年大阪大学基礎工学部情報科学科卒業．平成 23 年同大学大学院博士前期課程修了．在学時，コードクローンに関する研究に従事．



楠本 真二 （正会員）

昭 63 年大阪大学基礎工学部卒業．平成 3 年同大学大学院博士課程中退．同年同大学基礎工学部助手．平成 8 年同講師．平成 11 年同助教授．平成 14 年同大学大学院情報科学研究科助教授．平成 17 年同教授．博士（工学）．ソフトウェアの生産性や品質の定量的評価に関する研究に従事．電子情報通信学会，IEEE，IFPUG 各会員．